Tema 11: Aplicaciones de programación funcional

Informática (2012–13)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 11: Aplicaciones de programación funcional

Tema 11: Aplicaciones de programación funcional

Presentación del juego

- *Cifras y letras* es un programa de Canal Sur que incluye un juego numérico cuya esencia es la siguiente:

  Dada una sucesión de números naturales y un número objetivo, intentar construir una expresión cuyo valor es el objetivo combinando los números de la sucesión usando suma, resta, multiplicación, división y paréntesis. Cada número de la sucesión puede usarse como máximo una vez. Además, todos los números, incluyendo los resultados intermedios tienen que ser enteros positivos (1,2,3,...).

- Ejemplos
  - Dada la sucesión 1, 3, 7, 10, 25, 50 y el objetivo 765, una solución es (1+50)*(25−10).
  - Para el problema anterior, existen 780 soluciones.
  - Con la sucesión anterior y el objetivo 831, no hay solución.

Formalización del problema: Operaciones

- Las operaciones son sumar, restar, multiplicar o dividir.

```
data Op = Sum | Res | Mul | Div

instance Show Op where
   show Sum = "+"
   show Res = "-"
   show Mul = "*"
   show Div = "/"
```

- `ops` es la lista de las operaciones.

```
ops :: [Op]
ops = [Sum,Res,Mul,Div]
```

Operaciones válidas

- `(valida o x y)` se verifica si la operación o aplicada a los números naturales `x` e `y` da un número natural. Por ejemplo,

```
valida Res 5 3  ⇝  True
valida Res 3 5  ⇝  False
valida Div 6 3  ⇝  True
valida Div 6 4  ⇝  False
```

```
valida :: Op -> Int -> Int -> Bool
valida Sum _ _ = True
valida Res x y = x > y
valida Mul _ _ = True
valida Div x y = y /= 0 && x `mod` y == 0
```

Operaciones válidas

- `(valida o x y)` se verifica si la operación `o` aplicada a los números naturales `x` e `y` da un número natural. Por ejemplo,

```
valida Res 5 3  ⇝  True
valida Res 3 5  ⇝  False
valida Div 6 3  ⇝  True
valida Div 6 4  ⇝  False
```

```
valida :: Op -> Int -> Int -> Bool
valida Sum _ _ = True
valida Res x y = x > y
valida Mul _ _ = True
valida Div x y = y /= 0 && x `mod` y == 0
```

Aplicación de operaciones

- `(aplica o x y)` es el resultado de aplicar la operación `o` a los números naturales `x` e `y`. Por ejemplo,

```
aplica Sum 2 3  ⇝  5
aplica Div 6 3  ⇝  2
```

```
aplica :: Op -> Int -> Int -> Int
aplica Sum x y = x + y
aplica Res x y = x - y
aplica Mul x y = x * y
aplica Div x y = x `div` y
```

Aplicación de operaciones

- `(aplica o x y)` es el resultado de aplicar la operación `o` a los números naturales `x` e `y`. Por ejemplo,

```
aplica Sum 2 3  ⇝  5
aplica Div 6 3  ⇝  2
```

```
aplica :: Op -> Int -> Int -> Int
aplica Sum x y = x + y
aplica Res x y = x - y
aplica Mul x y = x * y
aplica Div x y = x `div` y
```

Expresiones

- Las expresiones son números enteros o aplicaciones de operaciones a dos expresiones.

```
data Expr = Num Int | Apl Op Expr Expr

instance Show Expr where
   show (Num n)     = show n
   show (Apl o i d) = parentesis i ++ show o ++ parentesis d
                      where
                         parentesis (Num n) = show n
                         parentesis e       = "(" ++ show e ++ ")"
```

- Ejemplo: Expresión correspondiente a (1+50)*(25−10)

```
ejExpr :: Expr
ejExpr =  Apl Mul e1 e2
    where e1 = Apl Sum (Num 1) (Num 50)
          e2 = Apl Res (Num 25) (Num 10)
```

Números de una expresión

- `(numeros e)` es la lista de los números que aparecen en la expresión e. Por ejemplo,

```
*Main> numeros (Apl Mul (Apl Sum (Num 2) (Num 3)) (Num 7))
[2,3,7]
```

```
numeros :: Expr -> [Int]
numeros (Num n)     = [n]
numeros (Apl _ l r) = numeros l ++ numeros r
```

Números de una expresión

- `(numeros e)` es la lista de los números que aparecen en la expresión e. Por ejemplo,

```
*Main> numeros (Apl Mul (Apl Sum (Num 2) (Num 3)) (Num 7))
[2,3,7]
```

```
numeros :: Expr -> [Int]
numeros (Num n)      = [n]
numeros (Apl _ l r) = numeros l ++ numeros r
```

Valor de una expresión

- `(valor e)` es la lista formada por el valor de la expresión e si todas las operaciones para calcular el valor de e son números positivos y la lista vacía en caso contrario. Por ejemplo,

```
valor (Apl Mul (Apl Sum (Num 2) (Num 3)) (Num 7)) ⇝ [35]
valor (Apl Res (Apl Sum (Num 2) (Num 3)) (Num 7)) ⇝ []
valor (Apl Sum (Apl Res (Num 2) (Num 3)) (Num 7)) ⇝ []
```

```
valor :: Expr -> [Int]
valor (Num n)     = [n | n > 0]
valor (Apl o i d) = [aplica o x y | x <- valor i
                                  , y <- valor d
                                  , valida o x y]
```

Valor de una expresión

- `(valor e)` es la lista formada por el valor de la expresión e si todas las operaciones para calcular el valor de e son números positivos y la lista vacía en caso contrario. Por ejemplo,

```
valor (Apl Mul (Apl Sum (Num 2) (Num 3)) (Num 7)) ⇝ [35]
valor (Apl Res (Apl Sum (Num 2) (Num 3)) (Num 7)) ⇝ []
valor (Apl Sum (Apl Res (Num 2) (Num 3)) (Num 7)) ⇝ []
```

```
valor :: Expr -> [Int]
valor (Num n)     = [n | n > 0]
valor (Apl o i d) = [aplica o x y | x <- valor i
                                  , y <- valor d
                                  , valida o x y]
```

Funciones combinatorias: Sublistas

- `(sublistas xs)` es la lista de las sublistas de `xs`. Por ejemplo,

```
*Main> sublistas "bc"
["","c","b","bc"]
*Main> sublistas "abc"
["","c","b","bc","a","ac","ab","abc"]
```

```
sublistas :: [a] -> [[a]]
sublistas []     = [[]]
sublistas (x:xs) = yss ++ map (x:) yss
    where yss = sublistas xs
```

Funciones combinatorias: Sublistas

- `(sublistas xs)` es la lista de las sublistas de `xs`. Por ejemplo,

```
*Main> sublistas "bc"
["","c","b","bc"]
*Main> sublistas "abc"
["","c","b","bc","a","ac","ab","abc"]
```

```
sublistas :: [a] -> [[a]]
sublistas []     = [[]]
sublistas (x:xs) = yss ++ map (x:) yss
    where yss = sublistas xs
```

Funciones combinatoria: Intercalado

- `(intercala x ys)` es la lista de las listas obtenidas intercalando `x` entre los elementos de `ys`. Por ejemplo,

```
intercala 'x' "bc"   ⇝  ["xbc","bxc","bcx"]
intercala 'x' "abc"  ⇝  ["xabc","axbc","abxc","abcx"]
```

```haskell
intercala :: a -> [a] -> [[a]]
intercala x []     = [[x]]
intercala x (y:ys) =
    (x:y:ys) : map (y:) (intercala x ys)
```

Funciones combinatoria: Intercalado

- `(intercala x ys)` es la lista de las listas obtenidas intercalando `x` entre los elementos de `ys`. Por ejemplo,

```
intercala 'x' "bc"   ⇝  ["xbc","bxc","bcx"]
intercala 'x' "abc"  ⇝  ["xabc","axbc","abxc","abcx"]
```

```
intercala :: a -> [a] -> [[a]]
intercala x []     = [[x]]
intercala x (y:ys) =
    (x:y:ys) : map (y:) (intercala x ys)
```

Funciones combinatoria: Permutaciones

- `(permutaciones xs)` es la lista de las permutaciones de `xs`. Por ejemplo,

```
*Main> permutaciones "bc"
["bc","cb"]
*Main> permutaciones "abc"
["abc","bac","bca","acb","cab","cba"]
```

```
permutaciones :: [a] -> [[a]]
permutaciones []     = [[]]
permutaciones (x:xs) =
    concat (map (intercala x) (permutaciones xs))
```

Funciones combinatoria: Permutaciones

- `(permutaciones xs)` es la lista de las permutaciones de `xs`. Por ejemplo,

```
*Main> permutaciones "bc"
["bc","cb"]
*Main> permutaciones "abc"
["abc","bac","bca","acb","cab","cba"]
```

```
permutaciones :: [a] -> [[a]]
permutaciones []     = [[]]
permutaciones (x:xs) =
    concat (map (intercala x) (permutaciones xs))
```

Funciones combinatoria: Elecciones

- `(elecciones xs)` es la lista formada por todas las sublistas de `xs` en cualquier orden. Por ejemplo,

```
*Main> elecciones "abc"
["","c","b","bc","cb","a","ac","ca","ab","ba",
 "abc","bac","bca","acb","cab","cba"]
```

```
elecciones :: [a] -> [[a]]
elecciones xs =
    concat (map permutaciones (sublistas xs))
```

Funciones combinatoria: Elecciones

- `(elecciones xs)` es la lista formada por todas las sublistas de `xs` en cualquier orden. Por ejemplo,

```
*Main> elecciones "abc"
["","c","b","bc","cb","a","ac","ca","ab","ba",
 "abc","bac","bca","acb","cab","cba"]
```

```
elecciones :: [a] -> [[a]]
elecciones xs =
    concat (map permutaciones (sublistas xs))
```

Reconocimiento de las soluciones

- `(solucion e ns n)` se verifica si la expresión `e` es una solución para la sucesión `ns` y objetivo `n`; es decir. si los números de `e` es una posible elección de `ns` y el valor de `e` es `n`. Por ejemplo,

```
solucion ejExpr [1,3,7,10,25,50] 765  =>  True
```

```
solucion :: Expr -> [Int] -> Int -> Bool
solucion e ns n =
    elem (numeros e) (elecciones ns) && valor e == [n]
```

Reconocimiento de las soluciones

- `(solucion e ns n)` se verifica si la expresión `e` es una solución para la sucesión `ns` y objetivo `n`; es decir. si los números de `e` es una posible elección de `ns` y el valor de `e` es `n`. Por ejemplo,

```
solucion ejExpr [1,3,7,10,25,50] 765  =>  True
```

```
solucion :: Expr -> [Int] -> Int -> Bool
solucion e ns n =
    elem (numeros e) (elecciones ns) && valor e == [n]
```

Tema 11: Aplicaciones de programación funcional

Divisiones de una lista

- `(divisiones xs)` es la lista de las divisiones de `xs` en dos listas no vacías. Por ejemplo,

```
*Main> divisiones "bcd"
[("b","cd"),("bc","d")]
*Main> divisiones "abcd"
[("a","bcd"),("ab","cd"),("abc","d")]
```

```
divisiones :: [a] -> [([a],[a])]
divisiones []     = []
divisiones [_]    = []
divisiones (x:xs) =
    ([x],xs) : [(x:is,ds) | (is,ds) <- divisiones xs]
```

Divisiones de una lista

- `(divisiones xs)` es la lista de las divisiones de `xs` en dos listas no vacías. Por ejemplo,

```
*Main> divisiones "bcd"
[("b","cd"),("bc","d")]
*Main> divisiones "abcd"
[("a","bcd"),("ab","cd"),("abc","d")]
```

```
divisiones :: [a] -> [([a],[a])]
divisiones []     = []
divisiones [_]    = []
divisiones (x:xs) =
    ([x],xs) : [(x:is,ds) | (is,ds) <- divisiones xs]
```

Expresiones construibles

- `(expresiones ns)` es la lista de todas las expresiones construibles a partir de la lista de números `ns`. Por ejemplo,

```
*Main> expresiones [2,3,5]
[2+(3+5),2-(3+5),2*(3+5),2/(3+5),2+(3-5),2-(3-5),
 2*(3-5),2/(3-5),2+(3*5),2-(3*5),2*(3*5),2/(3*5),
 2+(3/5),2-(3/5),2*(3/5),2/(3/5),(2+3)+5,(2+3)-5,
 ...
```

```
expresiones :: [Int] -> [Expr]
expresiones []  = []
expresiones [n] = [Num n]
expresiones ns  = [e | (is,ds) <- divisiones ns
                     , i       <- expresiones is
                     , d       <- expresiones ds
                     , e       <- combina i d]
```

Expresiones construibles

- `(expresiones ns)` es la lista de todas las expresiones construibles a partir de la lista de números `ns`. Por ejemplo,

```
*Main> expresiones [2,3,5]
[2+(3+5),2-(3+5),2*(3+5),2/(3+5),2+(3-5),2-(3-5),
 2*(3-5),2/(3-5),2+(3*5),2-(3*5),2*(3*5),2/(3*5),
 2+(3/5),2-(3/5),2*(3/5),2/(3/5),(2+3)+5,(2+3)-5,
 ...
```

```
expresiones :: [Int] -> [Expr]
expresiones []  = []
expresiones [n] = [Num n]
expresiones ns  = [e | (is,ds) <- divisiones ns
                     , i       <- expresiones is
                     , d       <- expresiones ds
                     , e       <- combina i d]
```

Combinación de expresiones

- `(combina e1 e2)` es la lista de las expresiones obtenidas combinando las expresiones e1 y e2 con una operación. Por ejemplo,

```
*Main> combina (Num 2) (Num 3)
[2+3,2-3,2*3,2/3]
```

```
combina :: Expr -> Expr -> [Expr]
combina e1 e2 = [Apl o e1 e2 | o <- ops]
```

Combinación de expresiones

- `(combina e1 e2)` es la lista de las expresiones obtenidas combinando las expresiones e1 y e2 con una operación. Por ejemplo,

```
*Main> combina (Num 2) (Num 3)
[2+3,2-3,2*3,2/3]
```

```
combina :: Expr -> Expr -> [Expr]
combina e1 e2 = [Apl o e1 e2 | o <- ops]
```

Búsqueda de las soluciones

- `(soluciones ns n)` es la lista de las soluciones para la sucesión `ns` y objetivo `n` calculadas por fuerza bruta. Por ejemplo,

```
*Main> soluciones [1,3,7,10,25,50] 765
[3*((7*(50-10))-25), ((7*(50-10))-25)*3, ...
*Main> length (soluciones [1,3,7,10,25,50] 765)
780
*Main> length (soluciones [1,3,7,10,25,50] 831)
0
```

```
soluciones :: [Int] -> Int -> [Expr]
soluciones ns n =  [e | ns' <- elecciones ns
                      , e   <- expresiones ns'
                      , valor e == [n]]
```

Búsqueda de las soluciones

- `(soluciones ns n)` es la lista de las soluciones para la sucesión `ns` y objetivo `n` calculadas por fuerza bruta. Por ejemplo,

```
*Main> soluciones [1,3,7,10,25,50] 765
[3*((7*(50-10))-25), ((7*(50-10))-25)*3, ...
*Main> length (soluciones [1,3,7,10,25,50] 765)
780
*Main> length (soluciones [1,3,7,10,25,50] 831)
0
```

```
soluciones :: [Int] -> Int -> [Expr]
soluciones ns n =  [e | ns' <- elecciones ns
                      , e   <- expresiones ns'
                      , valor e == [n]]
```

Estadísticas de la búsqueda por fuerza bruta

- Estadísticas:

```
*Main> :set +s
*Main> head (soluciones [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
(8.47 secs, 400306836 bytes)
*Main> length (soluciones [1,3,7,10,25,50] 765)
780
(997.76 secs, 47074239120 bytes)
*Main> length (soluciones [1,3,7,10,25,50] 831)
0
(1019.13 secs, 47074535420 bytes)
*Main> :unset +s
```

Tema 11: Aplicaciones de programación funcional

Resultados

- `Resultado` es el tipo de los pares formados por expresiones válidas y su valor.

```
type Resultado = (Expr,Int)
```

- `(resultados ns)` es la lista de todos los resultados construibles a partir de la lista de números `ns`. Por ejemplo,

```
*Main> resultados [2,3,5]
[(2+(3+5),10), (2*(3+5),16), (2+(3*5),17), (2*(3*5),30), ((2+3)+5,10),
 ((2+3)*5,25), ((2+3)/5,1),  ((2*3)+5,11), ((2*3)-5,1),  ((2*3)*5,30)]
```

```
resultados :: [Int] -> [Resultado]
resultados []  = []
resultados [n] = [(Num n,n) | n > 0]
resultados ns  = [res | (is,ds) <- divisiones ns
                      , ix      <- resultados is
                      , dy      <- resultados ds
                      , res     <- combina' ix dy]
```

Resultados

- `Resultado` es el tipo de los pares formados por expresiones válidas y su valor.

```
type Resultado = (Expr,Int)
```

- `(resultados ns)` es la lista de todos los resultados construibles a partir de la lista de números `ns`. Por ejemplo,

```
*Main> resultados [2,3,5]
[(2+(3+5),10), (2*(3+5),16), (2+(3*5),17), (2*(3*5),30), ((2+3)+5,10),
 ((2+3)*5,25), ((2+3)/5,1),  ((2*3)+5,11), ((2*3)-5,1),  ((2*3)*5,30)]
```

```
resultados :: [Int] -> [Resultado]
resultados []  = []
resultados [n] = [(Num n,n) | n > 0]
resultados ns  = [res | (is,ds) <- divisiones ns
                      , ix      <- resultados is
                      , dy      <- resultados ds
                      , res     <- combina' ix dy]
```

Combinación de resultados

- `(combina' r1 r2)` es la lista de los resultados obtenidos combinando los resultados `r1` y `r2` con una operación. Por ejemplo,

```
*Main> combina' (Num 2,2) (Num 3,3)
[(2+3,5),(2*3,6)]
*Main> combina' (Num 3,3) (Num 2,2)
[(3+2,5),(3-2,1),(3*2,6)]
*Main> combina' (Num 2,2) (Num 6,6)
[(2+6,8),(2*6,12)]
*Main> combina' (Num 6,6) (Num 2,2)
[(6+2,8),(6-2,4),(6*2,12),(6/2,3)]
```

```
combina' :: Resultado -> Resultado -> [Resultado]
combina' (i,x) (d,y) =
    [(Apl o i d, aplica o x y) | o <- ops
                               , valida o x y]
```

Búsqueda combinando generación y evaluación

- `(soluciones' ns n)` es la lista de las soluciones para la sucesión `ns` y objetivo `n` calculadas intercalando generación y evaluación. Por ejemplo,

```
*Main> head (soluciones' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
*Main> length (soluciones' [1,3,7,10,25,50] 765)
780
*Main> length (soluciones' [1,3,7,10,25,50] 831)
0
```

```
soluciones' :: [Int] -> Int -> [Expr]
soluciones' ns n = [e | ns'   <- elecciones ns
                      , (e,m) <- resultados ns'
                      , m == n]
```

Estadísticas de la búsqueda combinada

- Estadísticas:

```
*Main> head (soluciones' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
(0.81 secs, 38804220 bytes)
*Main> length (soluciones' [1,3,7,10,25,50] 765)
780
(60.73 secs, 2932314020 bytes)
*Main> length (soluciones' [1,3,7,10,25,50] 831)
0
(61.68 secs, 2932303088 bytes)
```

Tema 11: Aplicaciones de programación funcional

Aplicaciones válidas

- `(valida' o x y)` se verifica si la operación o aplicada a los números naturales `x` e `y` da un número natural, teniendo en cuenta las siguientes reducciones algebraicas

```
x + y = y + x
x * y = y * x
x * 1 = x
1 * y = y
x / 1 = x
```

```
valida' :: Op -> Int -> Int -> Bool
valida' Sum x y = x <= y
valida' Res x y = x > y
valida' Mul x y = x /= 1 && y /= 1 && x <= y
valida' Div x y = y /= 0 && y /= 1 && x `mod` y == 0
```

Resultados válidos construibles

- `(resultados' ns)` es la lista de todos los resultados válidos construibles a partir de la lista de números `ns`. Por ejemplo,

```
*Main> resultados' [5,3,2]
[(5-(3-2),4),((5-3)+2,4),((5-3)*2,4),((5-3)/2,1)]
```

```
resultados' :: [Int] -> [Resultado]
resultados' []  = []
resultados' [n] = [(Num n,n) | n > 0]
resultados' ns  = [res | (is,ds) <- divisiones ns
                       , ix      <- resultados' is
                       , dy      <- resultados' ds
                       , res     <- combina'' ix dy]
```

Combinación de resultados válidos

- `(combina" r1 r2)` es la lista de los resultados válidos obtenidos combinando los resultados `r1` y `r2` con una operación. Por ejemplo,

```
combina'' (Num 2,2) (Num 3,3)  =>  [(2+3,5),(2*3,6)]
combina'' (Num 3,3) (Num 2,2)  =>  [(3-2,1)]
combina'' (Num 2,2) (Num 6,6)  =>  [(2+6,8),(2*6,12)]
combina'' (Num 6,6) (Num 2,2)  =>  [(6-2,4),(6/2,3)]
```

```
combina'' :: Resultado -> Resultado -> [Resultado]
combina'' (i,x) (d,y) =
    [(Apl o i d, aplica o x y) | o <- ops
                               , valida' o x y]
```

Búsqueda mejorada mediante propiedades algebraicas

- `(soluciones" ns n)` es la lista de las soluciones para la sucesión `ns` y objetivo `n` calculadas intercalando generación y evaluación y usando las mejoras aritméticas. Por ejemplo,

```
*Main> head (soluciones'' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
*Main> length (soluciones'' [1,3,7,10,25,50] 765)
49
*Main> length (soluciones'' [1,3,7,10,25,50] 831)
0
```

```
soluciones'' :: [Int] -> Int -> [Expr]
soluciones'' ns n = [e | ns'   <- elecciones ns
                       , (e,m) <- resultados' ns'
                       , m == n]
```

Estadísticas de la búsqueda mejorada

- Estadísticas:

```
*Main> head (soluciones'' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
(0.40 secs, 16435156 bytes)
*Main> length (soluciones'' [1,3,7,10,25,50] 765)
49
(10.30 secs, 460253716 bytes)
*Main> length (soluciones'' [1,3,7,10,25,50] 831)
0
(10.26 secs, 460253908 bytes)§
```

Comparación de las búsquedas

Comparación de las búsquedad problema de dados [1,3,7,10,25,50] obtener 765.

- Búsqueda de la primera solución:

```
                +---------------------+
                | segs. | bytes       |
+---------------+-------+-------------+
| soluciones    | 8.47  | 400.306.836 |
| soluciones'   | 0.81  |  38.804.220 |
| soluciones''  | 0.40  |  16.435.156 |
+---------------+-------+-------------+
```

Comparación de las búsquedas

- Búsqueda de todas las soluciones:

	segs.	bytes
soluciones	997.76	47.074.239.120
soluciones'	60.73	2.932.314.020
soluciones''	10.30	460.253.716

Comparación de las búsquedas

Comprobación de que dados [1,3,7,10,25,50] no puede obtenerse 831

	segs.	bytes
soluciones	1019.13	47.074.535.420
soluciones'	61.68	2.932.303.088
soluciones''	10.26	460.253.908

El problema de las N reinas

- Enunciado: Colocar N reinas en un tablero rectangular de dimensiones N por N de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.
- El problema se representa en el módulo `Reinas`. Importa la diferencia de conjuntos `(\\)` del módulo `List`:

```
module Reinas where
import Data.List ((\\))
```

- El tablero se representa por una lista de números que indican las filas donde se han colocado las reinas. Por ejemplo, `[3,5]` indica que se han colocado las reinas `(1,3)` y `(2,5)`.

```
type Tablero = [Int]
```

El problema de las N reinas

- `reinas n` es la lista de soluciones del problema de las N reinas. Por ejemplo, `reinas 4` ⇝ `[[3,1,4,2],[2,4,1,3]]`. La primera solución `[3,1,4,2]` se interpreta como

	R		
			R
R			
		R	

```
reinas :: Int -> [Tablero]
reinas n = aux n
    where aux 0 = [[]]
          aux m = [r:rs | rs <- aux (m-1),
                          r <- ([1..n] \\ rs),
                          noAtaca r rs 1]
```

El problema de las N reinas

- `reinas n` es la lista de soluciones del problema de las N reinas. Por ejemplo, `reinas 4` ⇝ `[[3,1,4,2],[2,4,1,3]]`. La primera solución `[3,1,4,2]` se interpreta como

	R		
			R
R			
		R	

```
reinas :: Int -> [Tablero]
reinas n = aux n
    where aux 0 = [[]]
          aux m = [r:rs | rs <- aux (m-1),
                          r <- ([1..n] \\ rs),
                          noAtaca r rs 1]
```

El problema de las N reinas

- `noAtaca r rs d` se verifica si la reina `r` no ataca a niguna de las de la lista `rs` donde la primera de la lista está a una distancia horizontal `d`.

```
noAtaca :: Int -> Tablero -> Int -> Bool
noAtaca _ [] _ = True
noAtaca r (a:rs) distH = abs(r-a) /= distH &&
                         noAtaca r rs (distH+1)
```

El problema de las N reinas

- `noAtaca r rs d` se verifica si la reina `r` no ataca a niguna de las de la lista `rs` donde la primera de la lista está a una distancia horizontal `d`.

```
noAtaca :: Int -> Tablero -> Int -> Bool
noAtaca _ [] _ = True
noAtaca r (a:rs) distH = abs(r-a) /= distH &&
                         noAtaca r rs (distH+1)
```

Números de Hamming

- Enunciado: Los números de Hamming forman una sucesión estrictamente creciente de números que cumplen las siguientes condiciones:
  1. El número 1 está en la sucesión.
  2. Si x está en la sucesión, entonces $2x$, $3x$ y $5x$ también están.
  3. Ningún otro número está en la sucesión.
- `hamming` es la sucesión de Hamming. Por ejemplo,

```
take 12 hamming ⇝ [1,2,3,4,5,6,8,9,10,12,15,16]
```

```
hamming :: [Int]
hamming = 1 : mezcla3 [2*i | i <- hamming]
                      [3*i | i <- hamming]
                      [5*i | i <- hamming]
```

Números de Hamming

- Enunciado: Los números de Hamming forman una sucesión estrictamente creciente de números que cumplen las siguientes condiciones:
  1. El número 1 está en la sucesión.
  2. Si x está en la sucesión, entonces $2x$, $3x$ y $5x$ también están.
  3. Ningún otro número está en la sucesión.
- `hamming` es la sucesión de Hamming. Por ejemplo,

```
take 12 hamming ⇝ [1,2,3,4,5,6,8,9,10,12,15,16]
```

```
hamming :: [Int]
hamming = 1 : mezcla3 [2*i | i <- hamming]
                      [3*i | i <- hamming]
                      [5*i | i <- hamming]
```

Números de Hamming

- `mezcla3 xs ys zs` es la lista obtenida mezclando las listas ordenadas `xs`, `ys` y `zs` y eliminando los elementos duplicados. Por ejemplo,

```
Main> mezcla3 [2,4,6,8,10] [3,6,9,12] [5,10]
[2,3,4,5,6,8,9,10,12]
```

```
mezcla3 :: [Int] -> [Int] -> [Int] -> [Int]
mezcla3 xs ys zs = mezcla2 xs (mezcla2 ys zs)
```

Números de Hamming

- `mezcla3 xs ys zs` es la lista obtenida mezclando las listas ordenadas `xs`, `ys` y `zs` y eliminando los elementos duplicados. Por ejemplo,

```
Main> mezcla3 [2,4,6,8,10] [3,6,9,12] [5,10]
[2,3,4,5,6,8,9,10,12]
```

```
mezcla3 :: [Int] -> [Int] -> [Int] -> [Int]
mezcla3 xs ys zs = mezcla2 xs (mezcla2 ys zs)
```

Números de Hamming

- `mezcla2 xs ys zs` es la lista obtenida mezclando las listas ordenadas `xs` e `ys` y eliminando los elementos duplicados. Por ejemplo,

```
Main> mezcla2 [2,4,6,8,10,12] [3,6,9,12]
[2,3,4,6,8,9,10,12]
```

```
mezcla2 :: [Int] -> [Int] -> [Int]
mezcla2 p@(x:xs) q@(y:ys) | x < y     = x:mezcla2 xs q
                          | x > y     = y:mezcla2 p  ys
                          | otherwise = x:mezcla2 xs ys
mezcla2 []        ys                  = ys
mezcla2 xs        []                  = xs
```

Números de Hamming

- `mezcla2 xs ys zs` es la lista obtenida mezclando las listas ordenadas `xs` e `ys` y eliminando los elementos duplicados. Por ejemplo,

```
Main> mezcla2 [2,4,6,8,10,12] [3,6,9,12]
[2,3,4,6,8,9,10,12]
```

```
mezcla2 :: [Int] -> [Int] -> [Int]
mezcla2 p@(x:xs) q@(y:ys) | x < y     = x:mezcla2 xs q
                          | x > y     = y:mezcla2 p  ys
                          | otherwise = x:mezcla2 xs ys
mezcla2 []        ys                  = ys
mezcla2 xs        []                  = xs
```

Bibliografía

1. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 11: The countdown problem.
2. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 13: Puzzles y solitarios.